MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A.I. LABORATORY

Artificial Intelligence
Memo No. 396

December 1976
LOGO Memo No. 41.

Teaching the Computer to Add: An Example of Problem-Solving in an Anthropomorphic Computer Culture

by

Cynthia J. Solomon

Computers open up new ways to think about knowledge and learning. Learning computer science should draw upon and feed these new approaches. In a previous paper called "Leading a Child to a Computer Culture" I discuss some ways to do so in a very elementary context. This paper is a contribution to extending such thinking to a more advanced project.

The research described in this paper was conducted at Massachusetts Institute of Technology in the Artificial Intelligence Laboratory's LOGO Group under the support of the National Science Foundation Grant No. EC40708X and at Boston University.

Teaching the Computer to Add: An Example of Problem-Solving in an Anthropomorphic Computer Culture

by Cynthia J. Solomon

The project of defining numbers and setting up the rules of addition has been undertaken or discussed in elementary school math classes, in college mathematical logic classes, in computer science logic design, in systems programming courses and in child psychology courses, and so on. In each case very different aspects of the project have been stressed. Here, we look at the project as an example of problem solving in what we have called an Anthropomorphic Computer Culture.

This paper describes how to teach a computer to add numbers. "Teach?", you might say, "does that mean program?" Yes, the paper describes a programming project and how a student might develop it. So it gives a model for developing programs. But there is something else. Learning to add numbers is an experience we have all had. My model of developing the program uses ourselves as an anthropomorphic model for the computer (a useful resource for student programmers) and the computer as a model for us (a useful resource for everyone). The paper is really about ways to think about doing thse two things at once. Hence its unorthodox style. Much of it is in the form of a monolog which tries to reflect what goes on in my mind as I work on such a project; at least that part of what goes on which I would offer to a beginner as a model.

In elementary school kids are presented with "number facts" and "addition facts". But they are deprived of a most valuable and important notion: what it is like to make up a set of rules, to define a domain

under which these rules can be consistently applied. Kids are not given a chance to see the process at work, get a feel for the power of recursion, or get a sense of how procedures are built up, debugged, elaborated. The kids are not helped to look for tricks (or look at some "facts" or techniques as tricks towards making problems easier to deal with.) The idea that you can develop your own set of heuristics is a very important contribution to your own problem-solving abilities. Maybe an even more important idea is that of learning from mistakes or "bugs" and developing debugging techniques, but in a world where everything is a "fact" it is hard to appreciate or get involved with debugging processes.

This reaction might also be encountered in recursive function theory classes where the idea of recursion is known, but its real power as a problem solving instrument is not felt and where the Peano axioms are received as "facts" (albeit formal proofs are introduced.) Students come out of the experience as if they had undergone a set of well proscribed exercises. They think of recursion and inductive methods as hocus pocus, not something that is really practical, but something to be applied in very special situations like courses in recursive function theory or induction! They do not see the deep implications which this rich project offers them.

Of course, the case might be made for the overwhelming difficulty in giving kids a taste for debugging, heuristics, process, procedure without the use of computers. But in many computer courses, this project is again viewed very narrowly as an exercise, not as a way of gaining insights into the nature of intelligence.

We present a different point of view; here, in this computer

culture, we explore this project as a legitimate research question. We will build our own system, find our own way. We might draw upon personal knowledge of number facts and skill algorithms.

A View of the Computer Culture

Computers open up new ways to learn about the development of knowledge and thinking. The computerist--the computer scientist-teacher-mathematician-psychologist--can create a culture in which it is possible to observe students engaged in a learning process. We can experiment with teaching techniques and content areas. And thus, we can develop a computer culture conducive to learning for a range of students from naive to expert. So we take a bare computer and enrich it with languages to talk in and attach devices like turtles and music boxes so we have concrete things to do and talk with.

One reason turtles were introduced into this culture was to concretize an underlying heuristic principle in problem-solving--anthropomorphize! Make the idea come alive, be someone--albeit it lives only in the mind. Talking to inanimate objects and thus giving them life is an implicit pattern in our lives; we have tried to turn it to advantage and make it an explicit process. The turtle world is one example and easily fosters the idea of developing mental imagery for concretizing abstractions. But throughout this culture anthropomorphisms abound; we see the computer, the program, the debugging process, etc. as people we can talk to and talk about. We extend this even further to imagine "little men" residing in the computer and coming alive to carry out a procedure, then disappearing.

LOGO, the programming language we use, is designed to encourage anthropomorphisms. LOGO is procedural and recursive. The importance of having a procedural language is born out by our hypothesis that an essential aspect of the growth of our knowledge base is the process of absorbing local procedures into a hierarchical structure where only the top-most procedure is even recalled by name or description.

We observe the development of our programs from one buggy state to another. We feel ourselves learning from these bugs as we carefully modify the procedures so that their behavior grows closer to our goal state. As our experience increases we see that some bugs afford us brilliant new insights into unexpected ways of achieving results. We begin to watch for them, ready to capitalize on bugs. Bugs are living creatures which we name, pamper, scold, laugh at, laugh with, and learn from and enjoy.

Another anthropomorphic influence on the programming language design affected procedure names and variable names. Their composition has few restrictions and their size can be longer than most people want to type. In this anthropomorphic computer culture naming is an important element. It helps to separate out and identify one procedure from another and one bug from another. Again, the explicit use of naming as a problem-solving tool is recognized as an essential ingredient. A first step toward creative development for beginners occurs when they make up their first procedures and in so doing must give them names. This parallel activity can really be mind-blowing. The student "teaches a new word" to the computer. This power, to define and create, to give meanings to words, is reemphasized by the project described below. The same feeling of power

offered to beginners in their first experiences defining procedures is reinforced even more strongly when as more sophisticated students they create procedures to add numbers.

Discussing the Project

In the next sections a style of problem solving is presented in the context of developing an actual program for addition. The style is discursive and reflective as I attempt to follow through the project as if I were doing it now while talking to you.

The way the procedures are constructed in this paper reflects a definite style of problem-solving. Rather than making a formal plan first by flow charting for example, I rely on a procedural approach which is more natural and intuitive. Procedural thinking and in particular recursive thinking in themselves encourage a structuring and planning out. Advice like: Reduce the problem, simplify, do first what you know how to do, defer problems, try to limit the number of jobs any one procedure has to do so that its role is clear--is an active part of the procedural development of a program.

Here is an example of the kind of discussion which might occur withe students.

First of all let's remember we want to make up an addition operation so that we can say

```
PRINT ADD 16 532
```

and the computer will say

```
548.
```

We also have to remember that there are no arithmetic operators (helpers) available to us.

How do computers really add? Some people answer: it's in their hardware; it's built into the system; it's hardwired. Is addition "hardwired" into our system, are we like computers and so if a wire is loose we can't do it.

It is true that arithmetic is a very necessary part of any computer's hardware, but the hardware is made up of "logical units" which are based on the same ideas we will investigate. "Well, could we do without arithmetic primitives?" To the beginner it really doesn't seem possible. It's like recursion you tell me addition is not primitive but emotionally it just seems unthinkable. What about addition in children. Is it really a primitive or are there pieces of knowledge which are acquired. Maybe we are so familiar with addition that we forget its components. Yes, addition is a familiar operation. But what if we had to tell a little man how to add? Where do we start? We might ask ourselves if we know of a similar experience. Hey, look what we have to do is "teach the computer" to add -- just like we might teach a person! Well now teachers teach kids to add, we were once those kids, how did we learn - can we give ourselves some tips (But I thought it was hardwired and teacher just..).

At this point in past discussions two suggestions emerge. Teachers say we have to teach the computer the "number facts" and computerists say we have to build a 10 x 10 table. Great, I say, a beginning. To the teachers I ask how do we teach the number facts and what

are they and how many of them are there. To computer students I ask is a 10 x 10 table large enough and how do we organize it. The teachers will face these issues too, after all making a table is a way of "teaching" number facts.

What kind of table and what are number facts. A table of the sums of the first 100 numbers is very limited and building a larger table is still very limited. Is that what I have in my head. Isn't there a key idea or two that I could build on without exhausting the computer's memory.

Is it the case that children learn "number facts" like 16 + 20 = 36 as a primitive notion or is there a more primitive idea underlying it all. What do kids learn about numbers. They learn about their relationship to each other. They learn to order them. Sesame Street teaches kids to count from 1 to 10 (and now to 20). Let's pick up on that and teach the compputer to count.

Counting by 1

A first description of a procedure for counting might look like the following command:

```
TO COUNTUP :NUMBER
10 PRINT ADD1 :NUMBER
END
```

What we have is a procedure requireing one input, a number. COUNTUP's job is to print the number following this input. To do this, we know, 1 must be added to :NUMBER. Of course, we don't yet know how to do that job. But we apply a powerful heuristic--we pretent we know so that we can describe how to count. That is we imagine we have all the procedures we need to do

the job. What we are doing now is getting a feel for what those needs are, then naming and listing them, and then putting their details aside until later.

Actually we can temporarily "cheat" and define ADD1 to be

```
SUM 1 :NUMBER
```

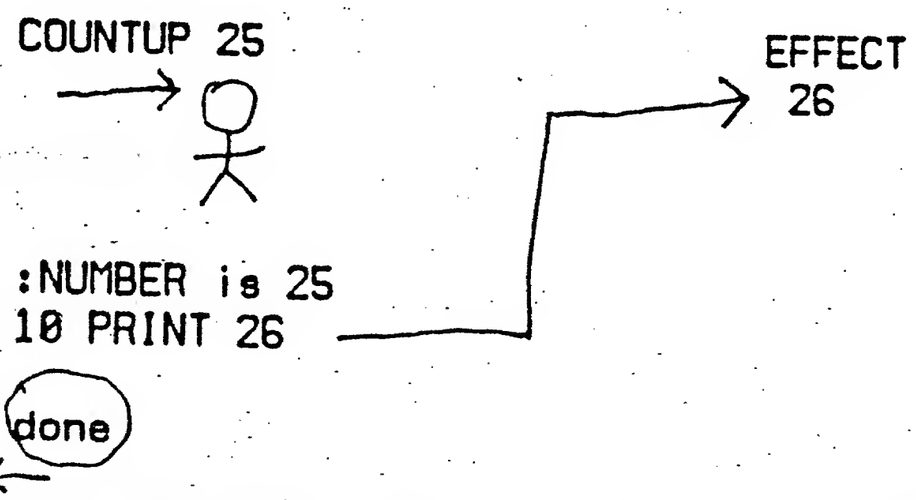
i.e.,

```
TO ADD1 :NUMBER
10 OUTPUT SUM 1 :NUMBER
END
```

We will replace this "phoney" algorithm later!!

DIGRESSION: In LOGO there are 2 procedure types: commands and operations. COUNTUP is a command, it does something but doesn't send back a message. ADD1 is an operation; it does send back a message.

Now we want to really understand what COUNTUP does. One method we offer is to trace through the script of a procedure in the guise of little men. Set into paper-and-pencil action a concrete example of COUNTUP at work. Thus



We embellish LOGO with a kind of meta-language in writing out what is happening. We use arrows to indicate flow.

Was this how you expected COUNTUP to behave? Not really. COUNTUP was supposed to continue counting. It was supposed to

```
COUNTUP ADD1 :NUMBER
```

and then COUNTUP ADD1 ADD1 :NUMBER

and then COUNTUP ADD1 ADD1 ADD1 :NUMBER

and so on.

We can look at the problem a little differently.

The action is to take :NUMBER and

PRINT ADD1 :NUMBER

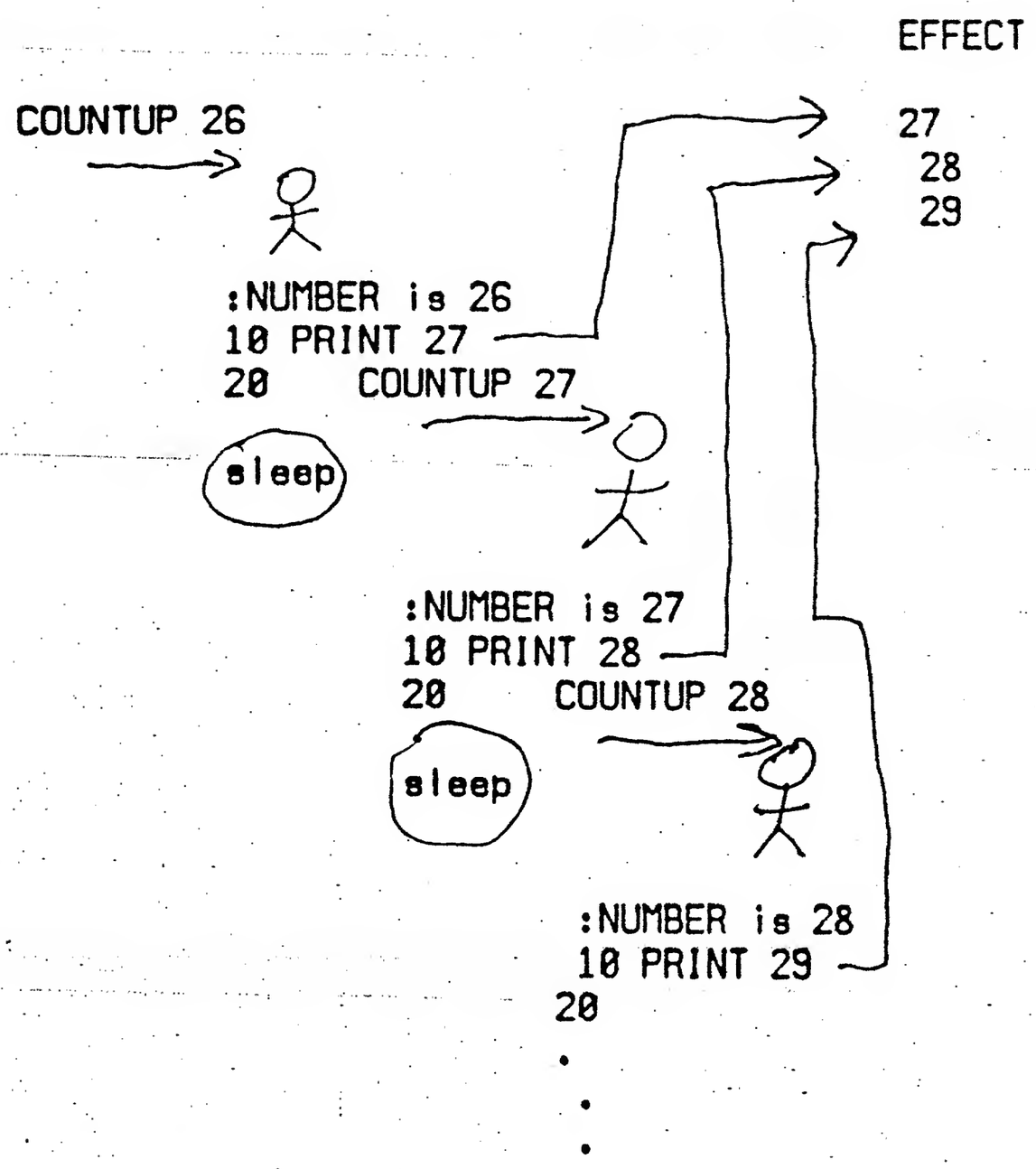
then we want that same action to be performed

on ADD1 :NUMBER and so on.

Well, let's tell COUNTUP to do just that. We change COUNTUP.

```
TO COUNTUP :NUMBER
10 PRINT ADD1 :NUMBER
20 COUNTUP ADD1 :NUMBER
END
```

COUNTUP tells itself to COUNTUP. Good. Let's trace through this version.



Okay, COUNTUP seems to be working well. It is a simple recursive procedure which doesn't know how to stop by itself. We see the little men never report back, they never disappear, but remain in a dormant state. Let's change the script by extending it to include a description of when the job is done and the process should be stopped.

To complete the description we have to give COUNTUP more information, another input. This second input could be an upper bound which :NUMBER must never exceed or it could be the number of times the process should be repeated starting from :NUMBER. The first way seems limiting and confusing in possible situations like

COUNTUP 19 17

where there would be no visible effect of COUNTUP doing anything. So we

follow the alternate suggestion.

Although we could change COUNTUP, let's not. Instead we will make a new procedure, REPEATADD1, and give it 2 inputs.

```
TO REPEATADD1 :NUMBER :TIMES
```

Its visible action is the same as COUNTUP's

```
PRINT ADD1 :NUMBER
```

except REPEATADD1 is going to stop on its own.

In deciding how to describe the STOP rule, we look at various possibilities. We could use a cute programming trick based on the "number fact" that if we reduce :TIMES by 1 each time the PRINT action is taken , :TIMES will eventually become 0. So we could

```
TEST :TIMES = 0
```

and

```
IFTRUE STOP
```

Oh, but we don't know how to reduce by 1. We don't know yet how to increase by 1!! So let's keep this possibility in mind for later on when we have plowed though the whole job. For now, let's look for another method, which will use only addition.

DIGRESSION: = is the infix form of EQUAL used here as a truth-valued identity operator not as a numeric "equals". By the way this special form of conditional is very useful froma pedagogic point of view because it is a more explicit statement of what is happening and easier to debug than

```
IF :TIMES = 0 STOP
```

Well, of course, we could conjure up a new specialist which couunts up from 0 (instead of down to 0) until it reaches :TIMES. Let's call the specialist "COUNTER. It can be a third input to REPEATADD1.

```
TO REPEATADD1 :NUMBER :TIMES :COUNTER
```

Then

```
10 TEST :TIMES = :COUNTER
20 IFTRUE STOP
```

otherwise

```
30 PRINT ADD1 :number
```

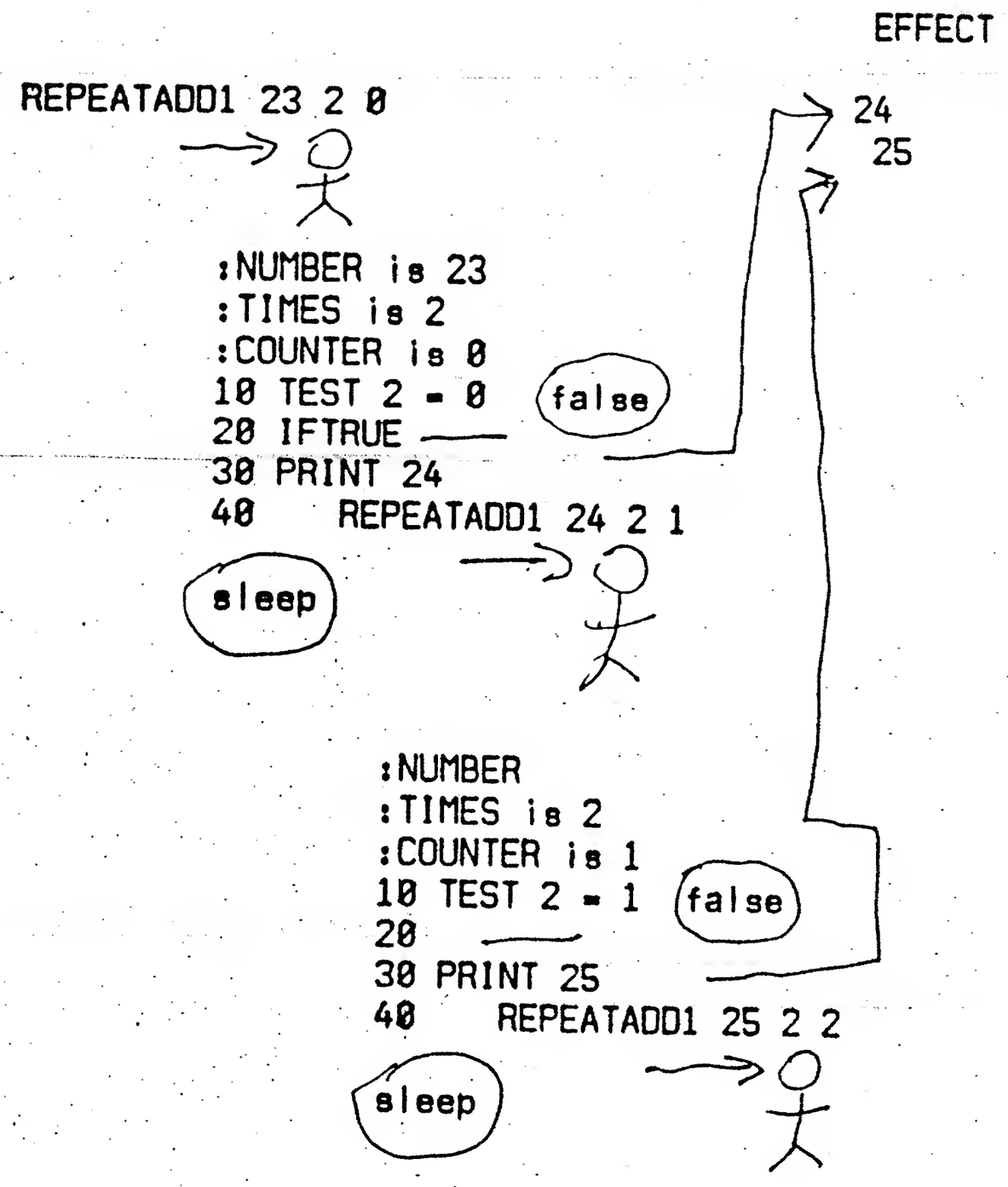
and now turn the job over to the next little man, but give him the changed inputs.

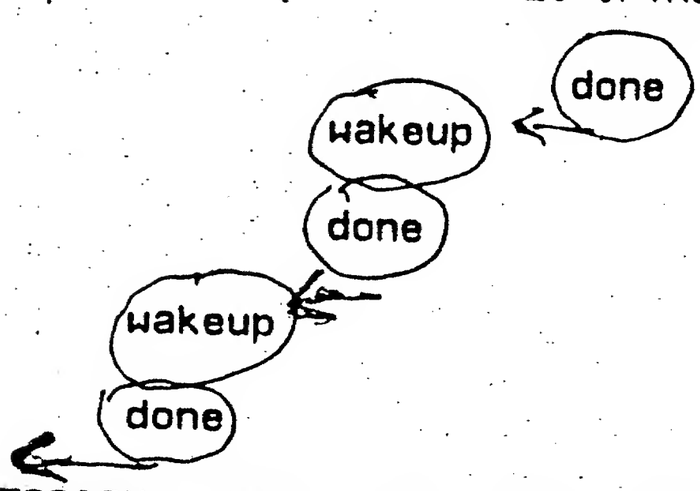
```
40 REPEATADD1
        ADD1 :NUMBER
        :TIMES
        ADD1 :COUNTER

END
```

DIGRESSION: A procedure with 3 inputs!! It's strikingly cumbersome. True enough and we will alleviate the situation by creating a superprocedure. But it is extremely important to see that there are 3 separate roles to the job. By naming them we can talk about them.

Now let's see our little men at work.





DIGRESSION: This kind of "playing computer" really helps in debugging and in general understanding the flow of a process. But it is personal, and you have to be the initiator to really appreciate the help

Now that REPEATADD1 works we can create a superprocedure to handle

:COUNTER. let's make COUNTBY1 serve that purpose.

```
TO COUNTBY1 :NUMBER :TIMES
10 REPEATADD1
        :NUMBER
        :TIMES
        0
END
```

Now we try it.

```
COUNTBY1 51 8
52
53
54
55
56
57
58
59
```

Great!!

Hey look, 51 + 8 = 59. We really have an adding machine!!

There is a problem. The job isn't really done. We really are only interested in the final number not the intermediate ones. We might not always want to print the number. We want to be free to decide what to do with the result each time we set the procedure in motion. Stated in LOGO terms we want COUNTBY1 to be an operation not a command. We want COUNTBY1 to send back a message.

```
TO COUNTBY1 :NUMBER :TIMES
10 OUTPUT REPEATADD1
                :NUMBER
                :TIMES
                0
END
```

But, of course, for this change to work REPEATADD1 must be transformed into an operation. Okay, let's do it.

Let's look at REPEATADD1 as it now is.

```
TO REPEATADD1 :NUMBER :TIMES :COUNTER
10 TEST :TIMES = :COUNTER
20 IFTRUE STOP
30 PRINT ADD1 :NUMBER
40 REPEATADD1
        ADD1 :NUMBER
        :TIMES
        ADD1 :COUNTER
   END
```

Let's go through the script and see what needs changing so that we can convert REPEATADD1 to an operation. There are 3 actions taken by REPEATADD1.

1. when :TIMES=:COUNTER the procedure halts
2. a number is printed
3. the job is repeated on new inputs

Let's decide what changes need to be made in each case.

1. Instead of only halting when the job is done we want to send back :NUMBER. So

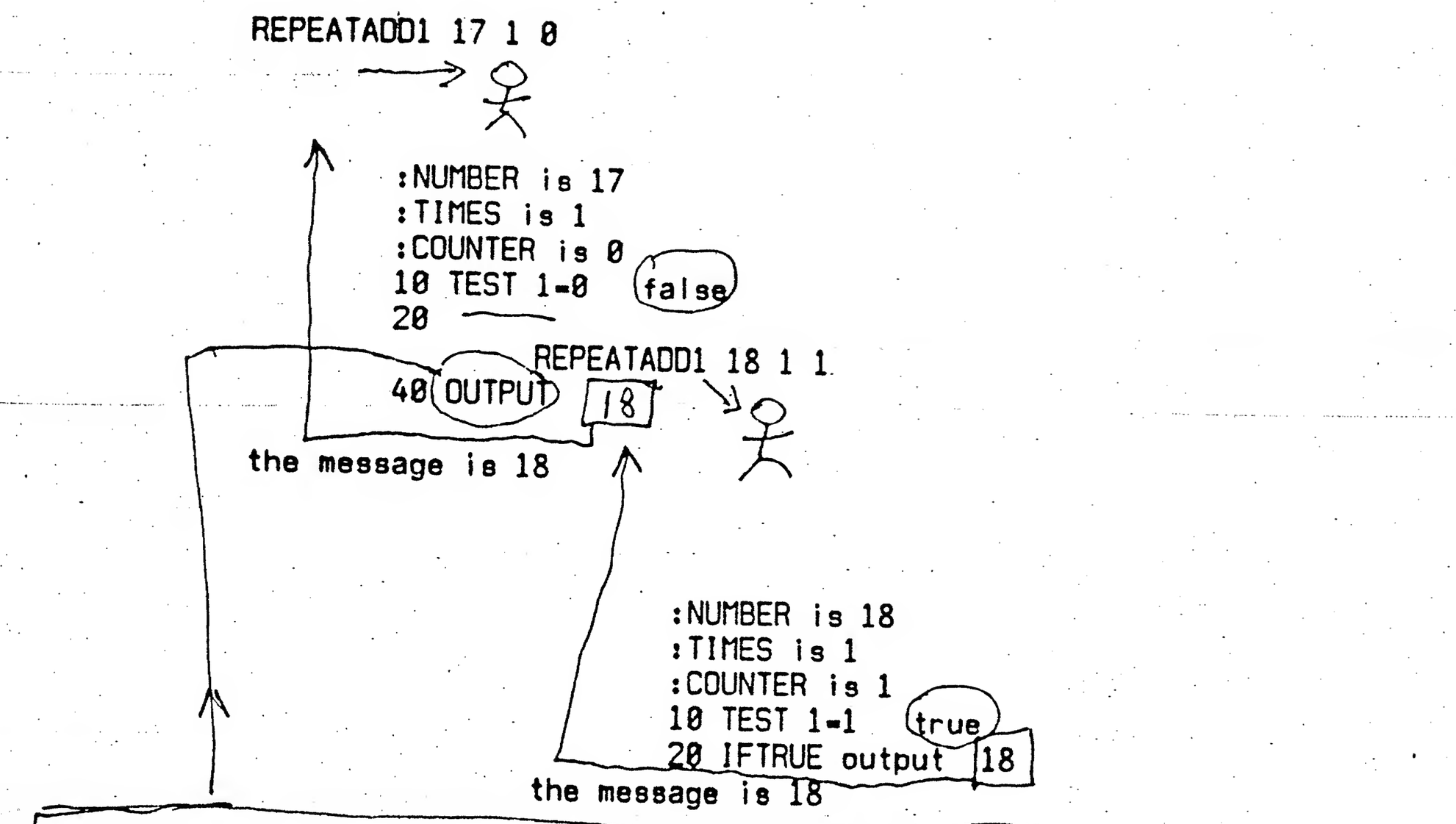
```
IFTRUE OUTPUT :NUMBER
```

2. We no longer want to print, so we can erase line 30.
3. In this case where the job is repeated but with new inputs, it is clear that the resultant action is what needs to be output. So

```
40 OUTPUT REPEATADD1
                ADD1 :NUMBER
                :TIMES
                ADD1 :COUNTER
```

This is obvious but also hard for many people to accept after a moment's reflection. Oh, the magic of recursion!! Well, it's not so magical. Put yourself into the place of little men. Play computer yourself.

For example, imagine you want to increase 17 by 1. So you give the job to REPEATADD1.



If there was no instruction telling what to do with the message a BUG would have occurred causing the computer to exclaim 'What do I do with 18'.

It looks like we have an operation which will do the job. Of course, the whole process depends upon ADD1. It's now time to look closely at what is needed there.

ADDing 1 to a Number

Let's add 1.

18 + 1 ---> 19

276 + 1 ---> 277

When we add 1 to a number we really transform the last digit of the number. So what we want is to take the number apart.

1 and 8

Then change 8 to 9.

Then put the new number together.

1 and 9

This should be easy. Let's call this input "NUMBER, so

```
WORD
  BUTLAST :NUMBER
  DIGITADD1 LAST :NUMBER
```

Thus

```
TO ADD1 :NUMBER
10 OUTPUT WORD
            BUTLAST :NUMBER
            DIGITADD1 LAST :NUMBER
END
```

Notice we have changed the problem to one which involves only digits--10 elements.

Adding 1 to a digit is simple. If the digit is 6, then the result is 7. If the digit is 0, then the result is 1. So we merely follow through on this idea and we have a procedure.

```
TO DIGITADD1 :DIGIT
10 TEST :DIGIT = 0
20 IFTRUE OUTPUT 1
30 IF :DIGIT = 5 OUTPUT 6
50 IF :DIGIT=9 OUTPUT 10
          .
          .
          .
```

DIGRESSION: Notice we have to check for each digit and so it doesn't matter in what order we check on the input's identity. Not does it matter about the form of conditional.

```
ADD1 20 ---> 21
ADD1 346 ---> 347
```

It looks like this procedure is working!!

Let's try countby1.

```
COUNTBY1 8 7 ---> 15

COUNTBY1 18 7 ---> 115
```

huh ... what's this!

COUNTBY1 176 5 ---> 1711

Oh, no! We have a bug!! Look

COUNTBY1 176 5

should really be 181 not 1711. Oh, ha, look. It's a CARRY bug

1 71 1

should be 7+1, i.e., 8.

Okay, let's look closely at what ADD1 does when the last digit of its input is 9.

ADD1 9 ---> 10

That's okay.

ADD1 19 ---> 110

Ugh!!

But now we know the bug, we can find a cure. We have to take special action when the last digit is 9. So

TO ADD1 :NUMBER

10 TEST 9 = LAST :NUMBER

20 IFTRUE

What should be done? That's simple.

ADD 1 to BUTLAST :NUMBER

and join that to 0 in place of LAST :NUMBER

```
20 IFTRUE OUTPUT WORD
                    ADD1 BUTLAST :NUMBER
                    0
```

and

```
30 OUTPUT WORD
          BUTLAST :NUMBER
          DIGITADD1 LAST :NUMBER
       END
```

The ultimate test might be

```
ADD1 999 ---> 1000
```

SUPERADD

There is still a slight problem. Imagine we want to add 99 and 9999. We will need 1000 little men, all alive although in a dormant state. That's tooo much both time-wise and work-area-size-wise. To be practical we have to reduce the amount of work. We can do that by applying the same methods we used in ADD1. Change the problem to be addition of digits whose results get concatenated together. So

```
TO ADD :N1 :N2
```

The p: count by one from LAST :N1; do it LAST :N2 times; do the same for the rest of the digits in :N1 and :N2; stick it all together.

```
WORD
          ADD BUTLAST :N1
              BUTLAST :N2
          COUNTBY1 LAST :N1
                   LAST :N2
```

Repeat this until either :N1 or :N2 is stripped of everything.

```
30 TEST EMPTYP :N2
40 IFTRUE OUTPUT :N1
50 OUTPUT WORD
        ADD BUTLAST :N1
            BUTLAST :N2
        COUNTBY1 LAST :N1
                 LAST :N2
END
```

Extensions

The project is done in a sense, but it is not closed to extensions. For example, you could rewrite the procedures using SUBTRACT1 as well as ADD1. Better still, you could extend the domain to include negative numbers or even decimals. Another direction to take is to make up other arithmetic operations like MULTIPLY or DIVIDE or extend this to any base system or build a modular arithmetic system, etc.

Are there some number operations which must be built into the programming language at a more primitive level? The one that comes immediately to mind is CLOCK, an operation which reports on the ticks of the computers clock. What about operations not restricted to numbers? Which are really primitive?